# CS 124 Programming Assignment 2: Spring 2022

**Your name(s) (up to two):** Burak Ufuktepe

**Collaborators:** None

No. of late days used on previous psets: 10

No. of late days used after including this pset:

## Analytical Approach

Let and be the time required to multiply two matrices using the conventional matrix multiplication algorithm and Strassen’s algorithm, respectively.

Let be the time required to multiply two matrices using the conventional matrix multiplication algorithm. The conventional matrix multiplication requires multiplications and additions for each element and there are elements in the matrix. Therefore, the runtime for the conventional matrix multiplication is:

Let be the time required to multiply two matrices using Strassen’s algorithm. Strassen’s algorithm performs 7 multiplications and 18 additions/subtractions of matrices of size . The reason why we use a ceiling function is because needs to be divided in half evenly, if is odd we would need to pad the matrix with one extra zero row and one extra zero column. Thus, we can express the recurrence relation as follows:

To find the cross-over point we need to find the value for when . So, we can simply use the equality and solve for .

If is even, we have:

This shows when is even and less than 15, switching to the conventional algorithm will be more optimal than using Strassen’s algorithm.

If is odd, we will have to pad the matrices with one extra zero row and one extra zero column. Let . After padding the matrices and dividing by half we obtain:

So we have:

This shows when is odd and less than 37.2, switching to the conventional algorithm will be more optimal than using Strassen’s algorithm.

## Implementation

### Data Layout Optimization

Splitting matrices takes up a significant portion of the actual runtime in Strassen’s algorithm. To speed up this process, instead of using a standard row-major ordering, Morton ordering is used to represent matrices. Morton ordering takes a 2D array stored in row-major order and arranges the matrix in block arrays where is the size of the “base case”. This is illustrated in Figure 1 for and Figure 2 for .

Table

Description automatically generated

Figure 1: Row-Major Ordering vs Morton Ordering (2 x 2 blocks)

Diagram

Description automatically generated with medium confidence

Figure 2: Row-Major Ordering vs Morton Ordering (3 x 3 blocks)

This way we end up with a 1D array and each quadrant is stored contiguously in memory as shown in Figure 3. To partition an matrix into four matrices we can simply slice the 1D array into four parts without having to iterate over each element.

Text

Description automatically generated with low confidence

Figure 3: 1D Array Representation of a Matrix using Morton Ordering

To compare the performance of Morton Ordering to Row-Major Ordering, several trials are conducted using random matrices where each entry is randomly selected to be 0, 1 or 2. Matrices of sizes and are tested using various cross-over points. Test results are given in Table 1. Furthermore, for each matrix size, runtime vs cross-over point plots are shown in Figure 4, Figure 5, and Figure 6 where the minimum runtimes are highlighted with red dots.

Table 1: Morton-Ordering vs Row-Major Ordering Runtime Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **Cross-Over Point** | **Runtime (s)** | |
| **Morton-Ordering** | **Row-Major Ordering** |
| 1536 | 6 | 243 | 337 |
| 1536 | 12 | 187 | 236 |
| 1536 | 24 | **176** | 202 |
| 1536 | 48 | 181 | **196** |
| 1536 | 96 | 192 | 202 |
| 1792 | 7 | 347 | 467 |
| 1792 | 14 | 278 | 343 |
| 1792 | 28 | **273** | 308 |
| 1792 | 56 | 277 | **303** |
| 1792 | 112 | 288 | 317 |
| 2048 | 4 | 704 | 1036 |
| 2048 | 8 | 475 | 624 |
| 2048 | 16 | **393** | 470 |
| 2048 | 32 | 398 | **422** |
| 2048 | 64 | 414 | 445 |
| 2048 | 128 | 451 | 468 |

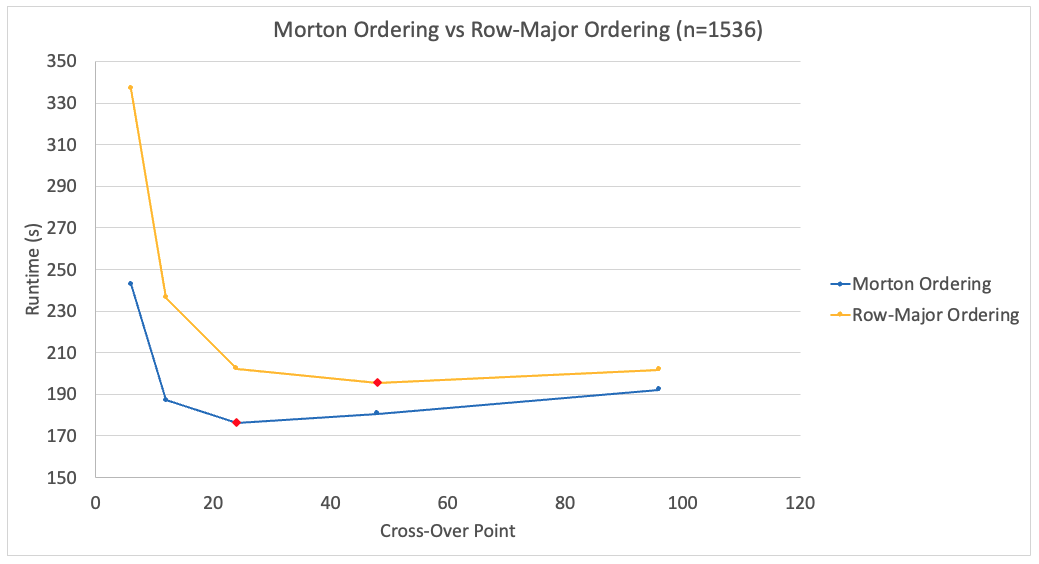


Figure 4: Morton Ordering vs Row-Major Ordering (n=1536)

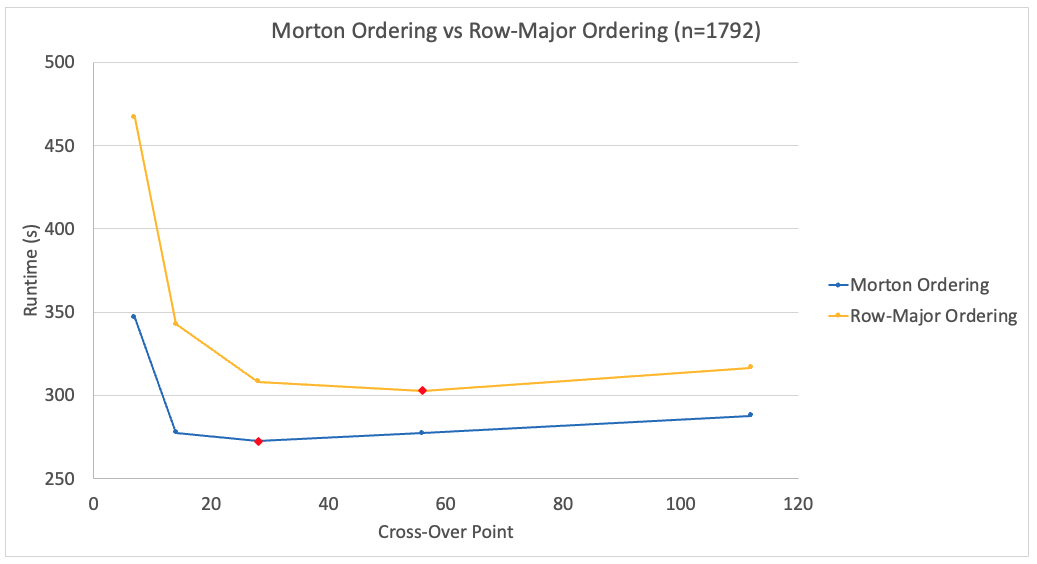


Figure 5: Morton Ordering vs Row-Major Ordering (n=1792)

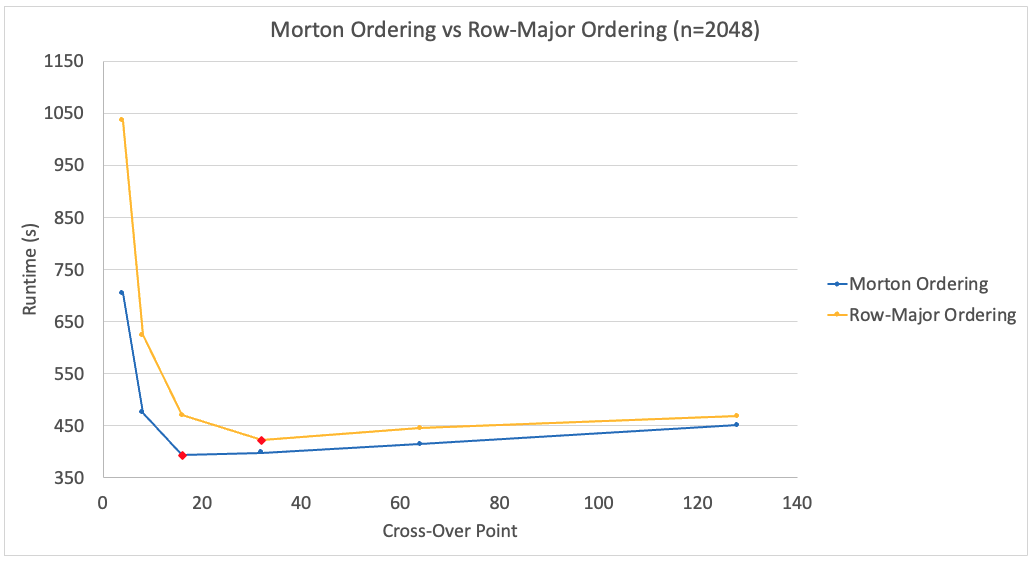


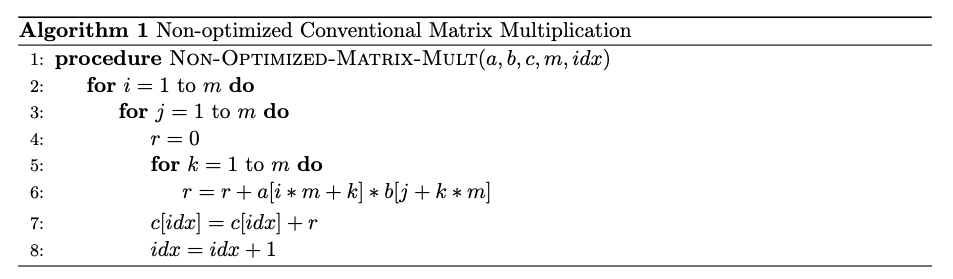
Figure 6: Morton Ordering vs Row-Major Ordering (n=2048)

Morton Ordering shows a noticeable performance gain over Row-Major Ordering (up to 32%). The largest performance gain is observed when the cross-over point is small. This is because using a smaller cross-over point results in more recursive calls in Strassen’s algorithm, as a result the number of matrix partition operations increases. Since splitting up a matrix in Morton Ordering is a lot cheaper compared to Row-Major Ordering, Morton Ordering yields better results in terms of runtime.

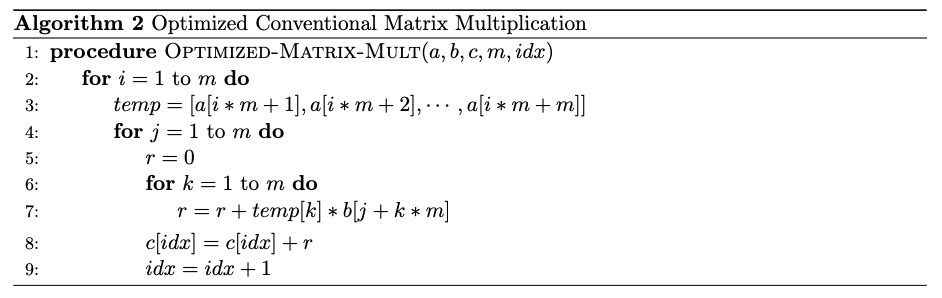
Also note that the optimum cross-over point for Morton Ordering is lower than the optimum cross-over point for Row-Major Ordering. This is because Strassen’s algorithm runs more efficiently when the matrices are laid out in Morton Ordering.

### Conventional Matrix Multiplication Optimization

In order to multiply two matrices that are Morton-ordered, the following procedure is used where and are the matrices to be multiplied, is the resultant Morton-ordered matrix that is initially filled with zeros, is the size of the matrix, and is the current index of that will be populated. Since this function is part of a recursive function is used as an input. As you can see in the for loop of lines 3-8, the procedure accesses multiple times and each time it performs a computation to find the relevant index of . Furthermore, due to Morton ordering, the procedure jumps between contiguous blocks of memory which slows down the process.



To optimize this procedure, the frequently accessed elements of are stored in a temporary array as shown in line 3 of the following procedure. Then, in line 7, this array is used to access the relevant element of . Since is a much smaller array it provides better cache performance since there is index locality and there is no computation involved to find the index.



Several trials have been conducted to compare the runtimes of these two procedures. Random matrices of sizes from up to are used and the results are given in Figure 5. It is observed that the optimized procedure runs 21% to 24% faster than the non-optimized procedure.

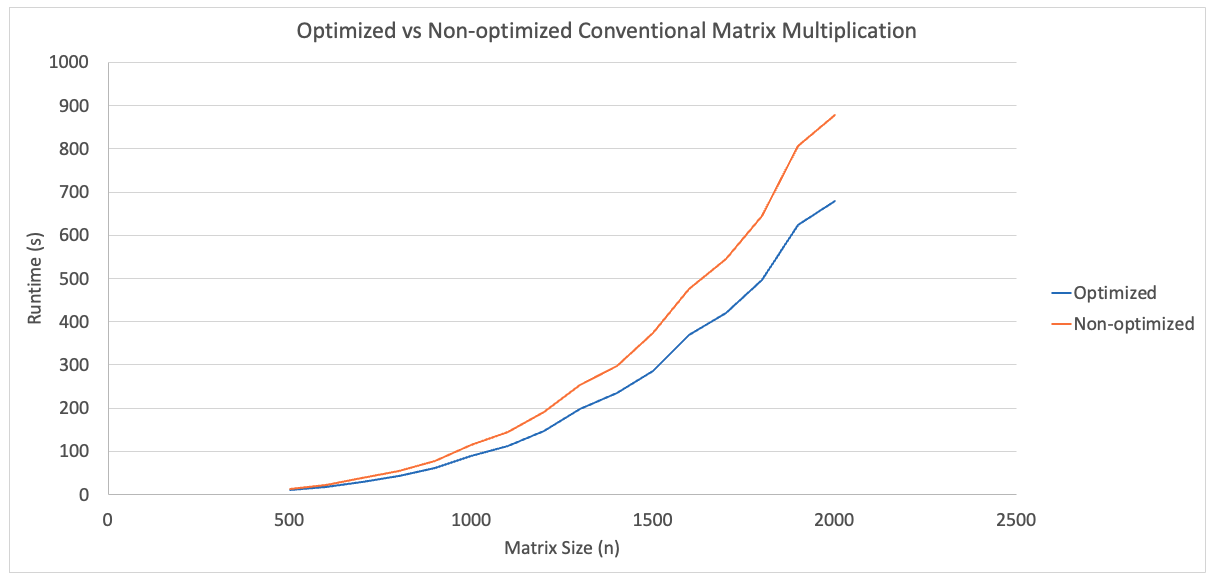


Figure 7: Optimized vs Non-optimized Conventional Matrix Multiplication

### Padding

Strassen’s algorithm recursively divides matrices into four matrices. That means at each recursive call must be divisible by 2. The obvious method to resolve this issue would be to pad the original matrix to the next power of two with zero rows and zero columns. However, this would be an expensive approach because we would have to almost double the size of the original matrix if its size is just over a power of 2 (i.e ).

Instead, we use the following approach: we first find the size of the “base case” for our Morton-ordered array. Then we keep doubling that size until we reach or exceed the size of our original matrix to find the minimum required size that we need to perform Strassen’s algorithm. That is:

1. Let . Repeatedly divide in half, each time taking the ceiling, until is less than or equal to the cross-over point. This will be equal to the size of the “base case” for our Morton-ordered array.
2. Then repeatedly double until .
3. Pad the original matrix with zero rows and zero columns until we obtain a matrix of size .

In other words, we are padding the matrix just enough so that Strassen’s algorithm can reach the “base case”. Since the padding is done upfront, we do not worry about odd-sized matrices while performing Strassen’s algorithm.

For example, assume and the cross-over point is 37. We find the size of the base case as follows:

🡪 🡪 🡪 🡪

Then we repeatedly double this size until we reach or exceed to find the minimum required matrix size:

🡪 🡪 🡪 🡪

Finally, we pad our original matrix until we reach a size of .

## Experimental Analysis

An experimental analysis is performed to find the optimum cross-over point. As explained in the previous section, our implementation pads the given matrices before performing Strassen’s algorithm (if necessary). Therefore, the size of the matrices is always converted from to where is the size of the “base case” such that and is the smallest integer such that . It is observed that this conversion process (padding) takes only a fraction of the total runtime. Hence, to simplify the analysis, padding is avoided in the experiments by selecting matrices of sizes where is the cross-over point. The following 16 matrix sizes are considered in the experimental analysis:

768, 832, 896, 960, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2048, 2304, 2560, 2816.

For each size, different cross-over points are considered. A sample calculation is given in Table 2 for

Table 2: Cross-Over Points Considered for n=1536

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

The runtime vs cross-over point graphs are shown in Figure 8 for and in Figure 9 for where the minimum runtime for each matrix size is highlighted with a red dot. In both graphs we observe that, for every matrix size, the minimum cross-over point that is larger than yields the minimum runtime. More specifically, the optimal cross-over point varies in the range of to . Hence

and the next largest cross-over point is greater than (the smallest one is for and ). This shows our experimental analysis is in line with our analytical analysis. Furthermore, for every matrix size,

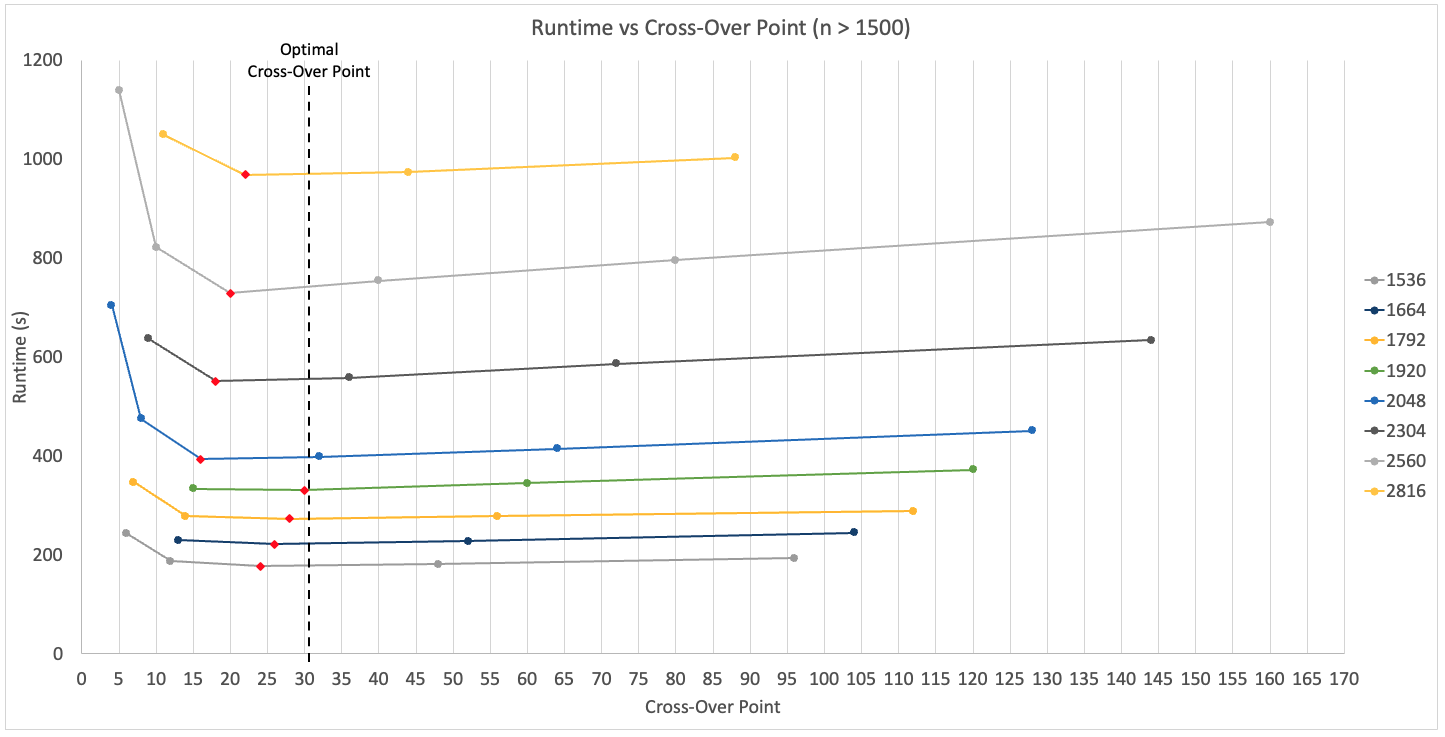


Figure 8: Runtime vs Cross-Over Point (n < 1500)

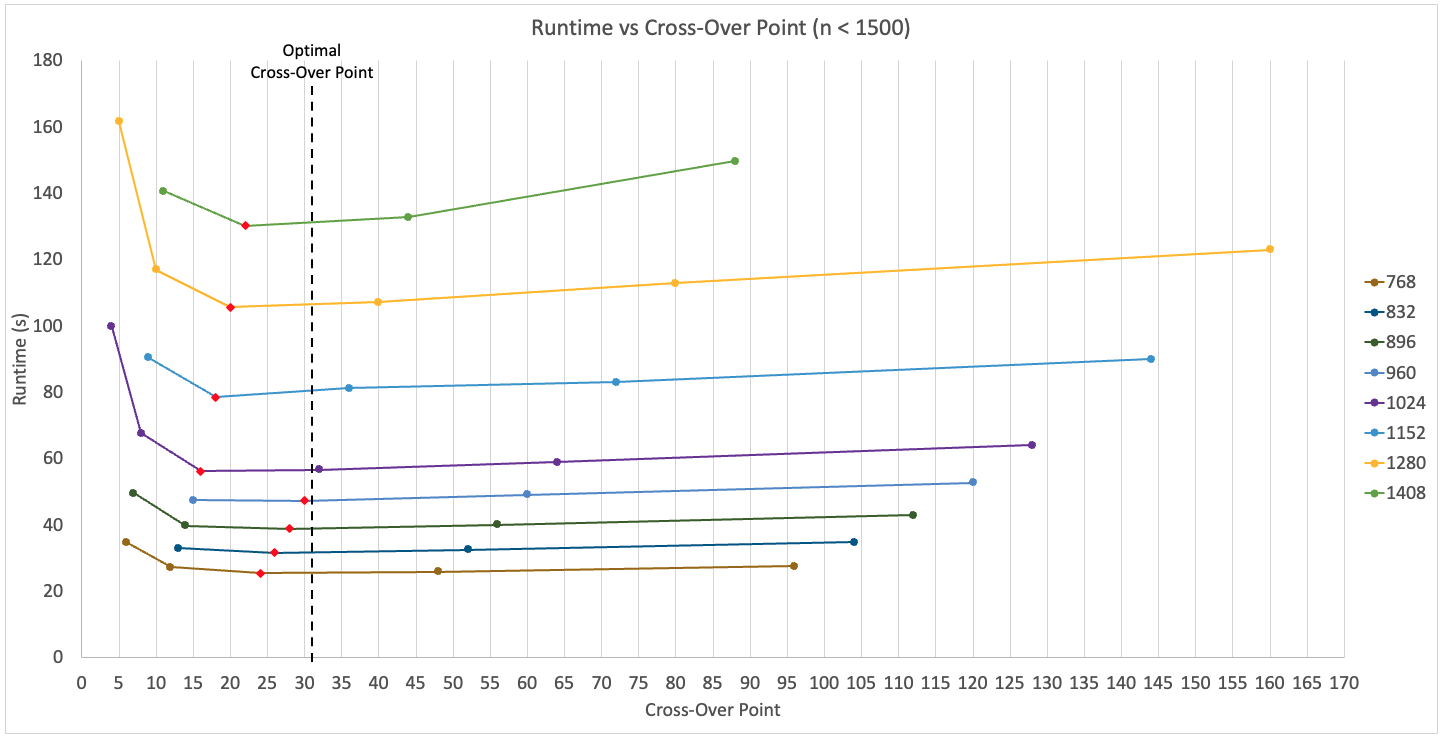


Figure 9: Runtime vs Cross-Over Point (n > 1500)